Making Mercury programs tail recursive

Extended abstract

Peter Ross, David Overton and Zoltan Somogyi
{petdr,dmo,zs}@cs.mu.oz.au
Department of Computer Science and Software Engineering
The University of Melbourne
Parkville, Victoria 3052, Australia

1 Introduction

Recursive calls are often very close to being the last action in a clause. Frequently, however, the recursive call computes a value which must be put into a memory cell or used in a simple computation. The current Mercury implementation cannot make such calls tail recursive. Therefore the generated code will consume stack space for each iteration, and will need to execute stack frame setup and teardown code in every iteration. This paper presents two optimisations that in many common cases can transform such code into a tail recursive form by moving code from after the recursive call to before the recursive call.

The first optimisation is a pure source to source transformation. It looks for recursive calls followed by associative computations which depend on the outputs of the recursive call. The optimisation then creates a specialised version of the predicate in which the associative computation is done before the recursive call. This specialised predicate will have extra arguments that act as *accumulators*: they hold the cumulative results of the associative computations so far.

The second optimisation looks for recursive calls followed by construction unifications that put some of the results of that call into memory cells, and moves those construction unifications before the call. This required extending the Mercury mode system to allow a restricted form of partially instantiated data structures and to record aliases, since in the transformed program the cell is not ground when it is created, but will become ground when the recursive call returns. To allow this to happen, we also had to extend the parameter passing conventions of the Mercury abstract machine to allow predicates to return selected output arguments in memory, not in registers.

Some predicates contain both associative computations and construction unifications after the recursive call. To allow us to make such predicates tail recursive, the accumulator introduction optimisation will perform its transformation even when the recursive call is followed by construction unifications as well as associative computations, knowing that the other optimisation will move the construction unifications before the call.

The rest of this paper is organised as follows. Section 2 describes the accumulator introduction transformation. Section 3 describes the changes needed to allow output arguments to be returned in memory. Section 4 discusses how our two optimisations are related to previous work. For space reasons, we cannot present an overview of the Mercury language; we refer the reader to the Mercury language reference manual [9].

2 Accumulator Introduction

This section introduces the source to source transformation that looks for self recursive predicates where the recursive call is followed by an associative computation. This associative computation can then be moved in front of the recursive call. This problem has previously been studied as a transformation based on unfold/fold rules [3, 7, 8] or as a schemata matching transformation [5, 6]. Our transformation is loosely based on Debray's [7] transformation.

2.1 Transformation

Consider a predicate whose body consists of a switch with two arms, where one arm of the switch contains the recursive call. We would like to determine whether or not the computation after the recursive call is associative. This is the purpose of the first stage of the transformation.

An associative predicate is one whose arguments can be divided into a (possibly empty) set of invariant inputs and three nonempty sets, two of inputs and one of outputs, for which the following law holds:

```
forall I, A, B, C, ABC:
    p(I, A, B, AB), p(I, AB, C, ABC) <=> p(I, B, C, BC), p(I, A, BC, ABC)
```

This law says that processing A, B and C from right to left produces the same value for ABC as processing them from left to right. One input for our algorithm is the list of predicates for which this law holds. Currently the predicates which are associative is hard-wired into the compiler. At a later date we would like to add the ability to annotate goals with laws like the above, and then for the compiler to infer associativity.

We illustrate how the accumulator introduction algorithm works by tracing the transformation of the predicate, `reverse_double`, in the mode `(in, out, out)`. The predicate, `reverse_double`, reverses a list of integers and also constructs a new list where each integer is doubled. Such predicates are frequently produced by deforestation [13]. The superhomogeneous form [11] of the code, as generated by the Mercury compiler except with meaningful variable names, is as follows, with the code a human programmer would write on the right. (Note that for unifications, we show mode information by writing `<=` for construction unifications, `=>` for deconstruction unifications, `==` for equality tests and `:=` for assignments.)

```
reverse_double(L, R, D) :-
    ( % switch on L
        L == [],
        R <= [],
        D <= []
    ;
        L => [H | T],
        reverse_double(T, R0, D0),
        Empty <= [],
        ListH <= [H | Empty],
        append(R0, ListH, R),
        Two <= 2,
        '*'(Two, H, Dbl),
        D <= [Dbl | D0]
    ).
```

```
reverse_double([], [], []).
reverse_double([H|T], R, D) :-
    reverse_double(T, R0, D0),
    append(R0, [H], R),
    D = [2*H | D0].
```

Construction unifications are not associative, so a simple associativity analysis might determine that the tail computation is non-associative. However, the call to `'*'` and all of the construction unifications except the last one can safely be moved before the recursive call without breaking the mode constraints of the procedure. This rearrangement of goals is trivial in the Mercury compiler, as at each point in a procedure it knows the state of instantiation of every variable, and two goals can have their order swapped providing the first goal doesn't modify the instantiation state of any variable that is used by the second goal.

The recursive case now ends in `reverse_double(T, R0, D0), append(R0, ListH, R), D <= [Dbl | D0]`. Since we will be rearranging the execution order inside this computation, we will introduce a new predicate, `reverse_double_acc`, with this body, whose arguments are the variables which occur both in this code fragment and in the rest of the code of `reverse_double_acc`:

```
reverse_double_acc(T, R, D, ListH, Dbl) :-
    reverse_double(T, R0, D0),
    append(R0, ListH, R),
    D <= [Dbl | D0].
```

We will fold the end of the recursive case of `reverse_double` with this new predicate. The compiler will be able to use last call optimisation on this call, and as we shall see, we can make `reverse_double_acc` tail recursive as well. This requires first unfolding the call to `reverse_double` in `reverse_double_acc` with the original definition of `reverse_double`. This of course requires a renaming apart step; we have put an `A_` prefix on the variables that came from `reverse_double_acc`.

```
reverse_double_acc(A_T, A_R, A_D, A_ListH, A_Dbl) :-
    ( % switch on A_T
        A_T == [],
        A_R0 <= [],
        A_D0 <= [],
        append(A_R0, A_ListH, A_R),
        A_D <= [A_Dbl | A_D0]
    ;
        A_T => [H | T],
        Empty <= [],
        ListH <= [H | Empty],
        Two <= 2,
        '*'(Two, H, Dbl),
        reverse_double(T, R0, D0),
        append(R0, ListH, A_R0),
        A_D0 <= [Dbl | D0],
        append(A_R0, A_ListH, A_R),
        A_D <= [A_Dbl | A_D0]
    ).
```

The second call to `append` doesn't depend on the preceding construction unification so their order can be swapped. Now by use of the associative law the two calls to `append` can be rearranged to become `append(ListH, A_ListH, New), append(R0, New, A_R)`, where `New` is a new variable of the same type as `ListH`, `A_ListH`, `R0` and `A_R`. (The associative law requires the types of those four variables to be the same.) The first call to `append` no longer depends on the recursive call, so it can be moved before the recursive call to give the following code:

```
reverse_double_acc(A_T, A_R, A_D, A_ListH, A_Dbl) :-
    ( % switch on A_T
        A_T == [],
        A_R0 <= [],
        A_D0 <= [],
        append(A_R0, A_ListH, A_R),
        A_D <= [A_Dbl | A_D0]
    ;
        A_T => [H | T],
        Empty <= [],
        ListH <= [H | Empty],
        Two <= 2,
        '*'(Two, H, Dbl),
        append(ListH, A_ListH, New),
        reverse_double(T, R0, D0),
        append(R0, New, A_R),
        A_D0 <= [Dbl | D0],
        A_D <= [A_Dbl | A_D0]
    ).
```

We now use the original definition of `reverse_double_acc` to do folding in both `reverse_double` and previous version of `reverse_double_acc` to obtain the following completely transformed code:

```
reverse_double(L, R, D) :-
    ( % switch on L
        L == [],
        R <= [],
        D <= []
    ;
        L => [H | T],
```

```
        Empty <= [],
        ListH <= [H | Empty],
        Two <= 2,
        '*'(Two, H, Dbl),
        reverse_double_acc(T, R, D, ListH, Dbl)
    ).

reverse_double_acc(A_T, A_R, A_D, A_ListH, A_Dbl) :-
    ( % switch on A_T
        A_T == [],
        A_R0 <= [],
        A_D0 <= [],
        append(A_R0, A_ListH, A_R),
        A_D <= [A_Dbl | A_D0]
    ;
        A_T => [H | T],
        Empty <= [],
        ListH <= [H | Empty],
        Two <= 2,
        '*'(Two, H, Dbl),
        append(ListH, A_ListH, New),
        reverse_double_acc(T, A_R, A_D0, New, Dbl),
        A_D <= [A_Dbl | A_D0]
    ).
```

The call to `append` in the base case is redundant. However, it is only redundant because for this particular predicate the base case initialises the answer to be the identity element for `append`. Partial deduction [10] can replace this call with `A_ListH = A_R`.

In the recursive case, application of the associativity law has rearranged the order of the arguments to `append`. The variable, `ListH`, is now placed at the start of the list, rather than the end. Thus the complexity of the call to `append` now becomes $O(1)$ rather than $O(N)$, improving the big O complexity of the algorithm as a whole. Unfortunately the complexity can also change in the reverse direction. To prevent this from happening we introduce a heuristic. The heuristic is based on the idea that the variable holding the accumulator is usually 'bigger' then the variable used to get the next result, thus it is this 'bigger' variable that contributes to the big O complexity of the predicate. For each associative predicate it is assumed that we know which argument positions are important for determining the complexity of a predicate. We then ensure that after rearrangement the accumulator variable is not in any of these positions. For `append` it is the first argument that is important for determining complexity, so the accumulator variable may not be placed there.

This optimisation is sufficient to introduce tail recursion in predicates in which the recursive call is followed by one or more independent associative computations. In its general form, which we will present in the full version of the paper, it also works when some of these computations depend on each other, provided that the dependent computations are commutative as well. For predicates such as `reverse_double` in which the code after the recursive call includes construction unification as well, achieving tail recursion also requires the optimisation described in section 3.

3 Last Call Modulo Constructor Application

The other tail recursion optimisation we have implemented is one that is very common in Prolog, but was much more difficult to implement in Mercury due to restrictions in the mode system and the argument passing conventions of the Mercury abstract machine. The recursive clause of `append(in, in, out)`,

```
append(Xs, Ys, Zs) :-
        Xs = [X | Xs1],
        Zs = [X | Zs0],
        append(Xs1, Ys, Zs0).
```

is tail recursive in Prolog due to *last call optimisation* [1]. However, in the Mercury abstract machine, all output values are returned in registers, so `Zs0` must be copied into memory after the recursive call has returned. Also, the current Mercury release does not allow the construction of partially instantiated terms, so the scheduler in the mode analyser will delay the entire construction of `[X | Zs0]` until after the recursive call.

We have added a new optimisation pass to the compiler which we call *last call modulo constructors* (LCMC).[1] This optimisation looks for predicates in which the last call is followed only by construction unifications and attempts to make the call into a tail call. It does this by moving the construction unifications to before the call and creating a new version of the called procedure that returns its output arguments in memory (i.e. pass-by-reference).

In order to support this optimisation, we extended the mode system and abstract machine to allow a limited form of partially instantiated data structures and to allow output arguments to be returned in memory.

In the Mercury mode system, the instantiation state (*inst*) of a variable is either `free`, `ground` or `bound(Functors)`, where `Functors` is the set of functors to which the variable may be bound and the insts of each argument of each functor. The *mode* of a predicate argument is a mapping from its initial inst to its final inst. The two most common argument modes, `ground -> ground` and `free -> ground`, are abbreviated `in` and `out`, respectively. The mode of a predicate (also known as a *procedure*) is a mode assignment for each argument of that predicate [9, 11].

Bromage [4] has recently extend the mode system to allow the tracking of definite aliases. His changes introduce a fourth alternative for insts, `alias(InstKey)`, where `InstKey` is an integer, and two variables are definitely aliased iff their insts are both `alias` specifying the same inst key. This allows the mode system to express that `Zs0` and the tail of `Zs` refer to the same storage location, which in turn allows the mode analyser to determine that when `Zs0` becomes ground, `Zs` also becomes ground.

In Bromage's system, an *inst key table* maps each inst key to a traditional inst, i.e. `ground`, `bound` or `free`. At the abstract machine level, a variable whose inst after expansion of aliases is `ground` or `bound` is represented by a memory location containing its value, while a variable whose expanded inst is `free` is represented by an uninitialised memory location. This is not sufficient for our needs. We have therefore added a new basic inst, `free_alias`, which is similar to `free` except that a variable whose expanded inst is `free_alias` is represented by a memory location that contains the address where its value will be placed when the variable is instantiated.

Since this representation allows only one memory reference per `free_alias` variable, it is not possible to unify two variables whose final insts are `free_alias`. This means that the LCMC optimisation can only be applied if each output from the recursive call is used in at most one field of one construction unification. Other representations, e.g. the one used in PARMA [12], would allow us to avoid this limitation. However, they would also be much more expensive, and we believe their cost would outweigh the benefits gained by LCMC.

An argument of a procedure is pass-by-reference if its mode is `free_alias -> ground`. The code generator will assign an input register to this argument and ensure that the address of the memory location where the result is required is placed in this register before calling the procedure. In the procedure itself, when the value of the argument is calculated, it will be placed in the specified memory location.

Let us look at how the transformation is applied to the procedure `append(in, in, out)` which we will refer to as mode 0 of the predicate `append/3`. The procedure is represented in the compiler's high level data structure (HLDS) in superhomogeneous form. The HLDS representation for this procedure, before LCMC, is shown below. The comments show the change in instantiation state of the variables after each sub-goal.

```
% mode 0 of append/3
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free
    (
```

[1] This name comes from a similar idea described by Wadler [14] which he calls *tail recursion modulo cons*.

```
        Xs == [],
        Zs := Ys                % Zs -> ground
    ;
        Xs => [X | Xs1],        % X -> ground, Xs1 -> ground
        append(Xs1, Ys, Zs0),   % call mode 0 of append/3
                                % Zs0 -> ground
        Zs <= [X | Zs0]         % Zs -> bound([ground | ground])
    ).
    % Zs -> ground
```

LCMC discovers that in the second arm of the switch, construction of `Zs` can be moved before the recursive call. `Zs0` now has an inst of `free_alias` before the call so LCMC tries to find a mode of `append/3` that matches this inst. The required mode is not found so it is created (mode 1) by copying mode 0 and changing the initial inst of `Zs`. At this stage, the HLDS for mode 0 is:

```
% mode 0 of append/3
:- mode append(in, in, out).

append(Xs, Ys, Zs) :-
    % Initial insts: Xs -> ground, Ys -> ground, Zs -> free
    (
        Xs == [],
        Zs := Ys                % Zs -> ground
    ;
        Xs => [X | Xs1]         % X -> ground,  Xs1 -> ground
        Zs <= [X | Zs0]         % Zs -> bound([ground | alias(IK0)])
                                % Zs0 -> alias(IK0)
                                % IK0 -> free_alias
        append(Xs1, Ys, Zs0)    % call mode 1 of append/3
                                % Zs -> bound([ground | alias(IK1)])
                                % Zs0 -> alias(IK1)
                                % IK1 -> ground
    ).
    % Zs -> ground
```

LCMC is now run on the new procedure, mode 1. When it re-orders the call and construction it finds that an appropriate mode of `append/3` already exists (i.e. mode 1). The HLDS for mode 1 looks the same as mode 0 except that the mode of the third argument is `free_alias -> ground` rather than `free -> ground` (`out`).

Finally, we show the C code generated by the compiler for the tail recursive mode 1 of `append`. (We have made the code more readable by e.g. shortening label names and removing type casts.) The arguments `Xs`, `Ys` and `Zs` are passed in registers `r1`, `r2` and `r3`, respectively.

```
/* code for predicate 'append'/3 in mode 1 */
Define_static(append_3_1);
    if (r1 == mkword(mktag(0), mkbody(0)))
        GOTO_LABEL(append_3_1_i2);
Define_label(append_3_1_i4);
    tag_incr_hp(r4, mktag(1), 2);
    field(mktag(1), r4, 0) = field(mktag(1), r1, 0);
    *r3 = r4;
    r3 = &field(mktag(1), r4, 1);
    r1 = field(mktag(1), r1, 1);
    if (r1 != mkword(mktag(0), mkbody(0)))
        GOTO_LABEL(append_3_1_i4);
Define_label(append_3_1_i2);
    *r3 = r2;
    proceed();
```

The code first checks whether `Xs` (in `r1`) is the empty list, and branches to label `append_3_1_i2` if it is. If it is not, it falls through to the code for the recursive case. This allocates a new two-word cell on the heap, and places a pointer to this cell tagged with the tag value for cons (1), which represents the value of `Zs`, in `r4`. It places the value of the `Zs` where the caller expects it, which is the memory location whose address the caller passed in `r3`. It fills in the first word of the new cell with `X`, and puts the address of the second into `r3`, to tell the recursive call where to put the value of `Zs0`. Extracting `Xs1` from the second field of the cons cell of `Xs` and putting it into `r1` completes the setup for the recursive call, since `Ys` is already in `r2`. The recursive call is an unconditional jump to the start of the procedure, which low-level optimisations in the Mercury compiler turn into the conditional jump to the start of the recursive case, as shown above. The code of the base case implements the assignment `Zs := Ys` by copying `Ys` to where the caller expects `Zs`.

The LCMC optimisation does not guarantee a tail call for every procedure it transforms. The call will be a tail call if and only if the candidate tail call returns exactly the same sequence of outputs in exactly the same locations as its containing procedure. If it does not, the call must be followed by code that places the output arguments of the containing procedure in their proper places. In such cases, the optimisation may still be useful as it may allow a tail call somewhere in the new pass-by-reference procedure by reducing the number of output registers of the procedure. This is seen in the `append` example where the call from mode 0 to mode 1 is not a tail call because mode 0 needs to save the value of `Zs` on the stack across the call and then place it in a register before it returns. However, the recursive call from mode 1 to itself is tail recursive because it can place `Zs` in the required location in memory as soon as it is constructed, so there is nothing that needs to be done after the recursive call.

Since the compiler runs LCMC recursively on each new procedure at the point where it is created, we can check whether the optimisation is actually going to give us a tail call somewhere in the new procedure and, if not, abandon the optimisation for this procedure.

The overall effect is that although our optimisation does not depend on the last call being a recursive call, the last call will be a recursive call in most of the situations in which our optimisation can be usefully applied.

The `reverse_double_acc/5` predicate that was created by the transformation in section 2 can also be transformed into tail recursive form by the LCMC transformation. Thus, the original `reverse_double/3` predicate, which could not be made tail recursive by either optimisation by itself, can be made tail recursive by combining the two optimisations.

4 Related Work

As stated earlier the accumulator introduction transformation was based on Debray's earlier work [7]. However Debray only showed how the transformation works with predicates which have the additional property of commutativity. Thus there is no need for the associative call to have its arguments rearranged, nor the introduction of the heuristic to ensure the big O complexity never increases. Brough and Hogger [5] allude to transformation not necessarily improving the efficiency, but only state that applicability of the transform should be related to the knowledge of the calling modes of each predicate.

The other significant facet of the accumulator introduction transformation is the possibility of leaving a predicate in a state which doesn't make the predicate tail recursive but allows the LCMC optimisation to be applied.

Optimisations similar to LCMC have often been suggested in the literature of both functional [14] and logic [11] programming. Most of these suggestions have not been implemented. The major exceptions are Prolog systems in which the issues involved in implementing LCMC are completely different.

One paper we know of which addresses the issue in the context of a moded logic programming language is Bigot and Debray [2]. They describe an algorithm for deciding when to return output values in registers and when to return them in memory. However, their technique requires profiling information (the number of times each procedure is called from each one of its call sites) which the Mercury profiler cannot yet supply, and in any case, their algorithm would require significant extensions to handle separate compilation. Our implementation of LCMC probably captures 95% of the benefit with much less implementation complexity.

5 Conclusion

We have presented two optimisations that can improve the operational efficiency of a program. Both optimisations can be automatically applied and have been implemented in the Melbourne Mercury compiler. We have also showed that both optimisations can be combined to optimise a program which neither optimisation would optimise by itself.

References

[1] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] Peter A. Bigot and Saumya Debray. Return value placement and tail call optimization in high level languages. *Journal of Logic Programming*, 38(1):1–29, January 1999.

[3] Charlene Bloch. Source-to-source transformation of logic programs. Master's thesis, Weizmann Institue of Science, November 1984.

[4] Andrew Bromage. A strong aliasing system. Master's thesis, Department of Computer Science and Software Engineering, University of Melbourne, Australia, to be submitted.

[5] D. R. Brough and C. J. Hogger. Compiling associativity into logic programs. *Journal of Logic Programming*, 4(4):345–359, December 1987.

[6] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9:115–134, 1991.

[7] Saumya K. Debray. Optimizing almost-tail-recursive Prolog programs. In *Proceedings of the International Symposium on Functional Programming Languages and Computer Architecture*, pages 204–219, 1985.

[8] Saumya K. Debray. Unfold/fold transformations and loop optimization of logic programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 297–307, 1988.

[9] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Australia, 1996.

[10] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.

[11] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[12] Andrew Taylor. *High Performance Prolog Implementation.* PhD thesis, University of Sydney, 1991.

[13] Simon Taylor. Optimization of Mercury programs. Honours report, Department of Computer Science, University of Melbourne, Australia, November 1998. `http://www.cs.mu.oz.au/mercury/information/papers.html`.

[14] P. Wadler. Listlessness is better than laziness: Lazy evaluation and garbage collection at compile-time. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 45–52, 1984.

6 Appendix A: Accumulator Introduction Algorithm

The initial stage of the algorithm partitions the goals following the final recursive call in the recursive case into four sets: the set of goals which can be safely moved before the recursive call (before), and three sets of goals which cannot, one of which (assoc) contains only calls to associative predicates while another (construct) contains only construction unifications. The algorithm goes on to the second stage only if the third such set (reject) is empty.

The second stage of the algorithm is responsible for ensuring that the transformation does not worsen the big-O computational complexity of the predicate, if it does worsen then the algorithm terminates. It also swaps the order of the arguments to all the associative calls and determines the substitutions that will be used to mimic the unfold/fold process that was used as the justification of the algorithm.

Finally the third stage creates the accumulator version of the predicate by applying the substitutions determined from stage 2. It also redefines the original predicate to call the accumulator version.

The following definitions are needed to allow the presentation of the algorithm to be as concise as possible. The operator, $\triangleright$, has the following meaning: $C_b \triangleright C_a$ is true iff C_b depends on C_a, which means that C_a changes the state of instantiation of some variable that occurs in C_b. This means that C_a and C_b must appear in the program in their original order; they cannot have their order swapped. The function $nonlocals(G)$ returns the set of variables of the goal, G, that are also used elsewhere in the predicate.

As an aid to understanding the algorithm, we will walk through the algorithm for the `reverse_double` predicate from section 2. First, we present the original code of the predicate annotated to show how the algorithm refers to its various parts:

```
reverse_double(L, R, D) :-  % Inv={} In={L} Out={R, D}
    ( % switch on L
        L => [H | T],                   % C 1,1
        reverse_double(T, R0, D0),      % C 1,2     In'={T} Out'={R0, D0}
        Empty <= [],                    % C 1,3
        ListH <= [H | Empty],           % C 1,4
        append(R0, ListH, R),           % C 1,5
        Two <= 2,                       % C 1,6
        '*'(Two, H, Dbl),               % C 1,7
        D <= [Dbl | D0]                 % C 1,8
    ;
        L == [],                        % C 2,1
        R <= [],                        % C 2,2
        D <= []                         % C 2,3
    ).
```

Note that the order of the arms of a switch does not matter, since exactly one of the arms will be taken. Note also that although in this example, all the conjuncts are calls or unifications, we do not require this in general; our algorithm would work just fine if e.g. $C_{1,6}$ were replaced with a compound goal (e.g. a disjunction).

We now present the values of the variables computed by our algorithm at the end of each stage of the transformation:

Stage 1:
$before = \{(1,1), (1,3), (1,4), (1,6), (1,7)\}$
$assoc = \{(1,5)\}$
$construct = \{(1,8)\}$

Stage 2:
$init\_accs = \{ListH, Dbl\}$
$acc\_var\_subst = \{ListH \rightarrow A\_ListH, Dbl \rightarrow A\_Dbl\}$
$inv\_input = \{\}$

$during\_assoc\_input = R0$
$before\_assoc\_input = ListH$
$assoc\_output = R$
$acc\_var = A\_ListH$
$CS_{1,5} = append(ListH, R0, R)$
$new\_acc = New$
$assoc\_call\_subst = \{R0 \to A\_ListH, R \to New\}$
$rec\_call\_subst = \{R0 \to R, ListH \to New\}$

Stage 3:
$call\_to\_head = \{R0 \to R, D0 \to D\}$
$head\_to\_call = \{R \to R0, D \to D0\}$
$p\_acc = reverse\_double\_acc$
$call = reverse\_double\_acc(T, R0, D0, ListH, Dbl)$
$base\_call = reverse\_double\_acc(T, R, D, ListH, Dbl)$
$rec\_call = reverse\_double\_acc(T, R, D0, New, Dbl)$
$R_{1,5} = append(ListH, A\_ListH, New)$
$R_{1,8} = D <= [A\_Dbl|D0]$
$B_{2,1} = L == []$
$B_{2,2} = R0 <= []$
$B_{2,3} = D0 <= []$
$B_{1,5} = append(R0, A\_ListH, R)$
$B_{1,8} = D <= [A\_Dbl|D0]$
$Acc = \{A\_ListH, A\_Dbl\}$

The final result of the algorithm are these two definitions:

```
reverse_double(L, R, D) :-
    ( % switch on L
        L => [H | T],                            % C 1,1
        Empty <= [],                             % C 1,3
        ListH <= [H | Empty],                    % C 1,4
        Two <= 2,                                % C 1,6
        '*'(Two, H, Dbl)                         % C 1,7
        reverse_double_acc(T, R, D, ListH, Dbl) % base_call
    ;
        L == [],                                 % C 2,1
        R <= [],                                 % C 2,2
        D <= []                                  % C 2,3
    ).
reverse_double_acc(L, R, D, A_ListH, A_Dbl) :-
    ( % switch on L
        L => [H | T],                            % C 1,1
        Empty <= [],                             % C 1,3
        ListH <= [H | Empty],                    % C 1,4
        Two <= 2,                                % C 1,6
        '*'(Two, H, Dbl)                         % C 1,7
        append(ListH, A_ListH, New),             % R 1,5
        reverse_double_acc(T, R, D0, New, Dbl), % rec_call
        D <= [A_Dbl | D0]                        % R 1,8
    ;
        L == [],                                 % B 2,1
        R0 <= [],                                % B 2,2
        D0 <= [],                                % B 2,3
        append(R0, A_ListH, R),                  % B 1,5
        D <= [A_Dbl | D0]                        % B 1,8
    ).
```

The code above for `reverse_double` is the same as what we showed in section 2. The code above for `reverse_double_acc` is also the same modulo renaming (the required substitution is A_T to L, A_R0 to R0, A_D0 to D0, A_R to R and A_D to D).

Let $p(\mathit{Inv}, \mathit{In}, \mathit{Out})$:-
  switch(
    $\langle C_{1,1} \ldots C_{1,k-1}, p(\mathit{Inv}, \mathit{In}', \mathit{Out}'), C_{1,k+1} \ldots C_{1,m}\rangle$,
    $\langle C_{2,1} \ldots C_{2,n}\rangle$
).
  where $1 \leq k < m \wedge \neg(\exists l, l > k \wedge C_{1,l} = p(\_))$

$\mathit{reject} = \mathit{construct} = \mathit{assoc} = \mathit{before} = \emptyset$
for $i = k+1$ to m:
  if $(C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1)\ldots(1,k-1)\} \cup \mathit{before}))$
    $\mathit{before} = \mathit{before} \cup (1,i)$
  else if ($C_{1,i}$ is an associative call such that
      (its invariant input arguments depend only on Inv
      $\wedge$ exactly one of its two associative input arguments occurs in Out')
      $\wedge C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1)\ldots(1,k)\} \cup \mathit{before})$)
    $\mathit{assoc} = \mathit{assoc} \cup (1,i)$
  else if ($C_{1,i}$ is a construction unification
      $\wedge C_{1,i} \triangleright C_{1,j} \Rightarrow (1,j) \in (\{(1,1)\ldots(1,k)\} \cup \mathit{before} \cup \mathit{construct})$)
    $\mathit{construct} = \mathit{construct} \cup (1,i)$
  else
    $\mathit{reject} = \mathit{reject} \cup (1,i)$
if $(\mathit{reject} \neq \emptyset)$
  transformation is not applicable, exit

Figure 1: The first stage

$before = before \cup \{(1,1)\ldots(1,k-1)\}$
$after = assoc \cup construct$
$init\_accs = \bigcup_{i \in before} nonlocals(C_i) \cap \bigcup_{i \in after} nonlocals(C_i)$

$acc\_var\_subst = \emptyset$
for each $var \in init\_accs$
    create a new variable, A_var
    $acc\_var\_subst = \{(var \rightarrow A\_var)\} \cup acc\_var\_subst$

$increase\_complexity = false$
$rec\_call\_subst = \emptyset$
$assoc\_call\_subst = \emptyset$
for each $a \in assoc$
    let inv_input
        be the set of invariant input arguments in C_a
    let $during\_assoc\_input$
        be the associative input argument in C_a that is in Out'
    let $before\_assoc\_input$
        be the associative input argument in C_a that is not in Out'
    let $assoc\_output$
        be the associative output argument in C_a

    let acc_var be the unique variable such that
        $(before\_assoc\_input \rightarrow acc\_var) \in acc\_var\_subst$

    let CS_a be C_a with $during\_assoc\_input$ and $before\_assoc\_input$
        having their positions swapped
    if ($during\_assoc\_input$ is in an inefficient position in CS_a)
        $increase\_complexity = true$

    create a new variable, new_acc
    $assoc\_call\_subst = \{during\_assoc\_input \rightarrow acc\_var,$
        $assoc\_output \rightarrow new\_acc\} \cup assoc\_call\_subst$
    $rec\_call\_subst = \{during\_assoc\_input \rightarrow assoc\_output,$
        $before\_assoc\_input \rightarrow new\_acc\} \cup rec\_call\_subst$
if ($increase\_complexity$)
    transformation would increase big-O complexity, exit

Figure 2: The second stage

$call\_to\_head = \emptyset$
$head\_to\_call = \emptyset$
for each $CV \in Out'$
    let HV be the variable from Out corresponding to CV in Out'
    $call\_to\_head = \{CV \rightarrow HV\} \cup call\_to\_head$
    $head\_to\_call = \{HV \rightarrow CV\} \cup head\_to\_call$

create new predicate, p_acc
$call = p\_acc(Inv, In', Out', init\_accs)$
$base\_call = rename\_goal(call, call\_to\_head)$
$rec\_call = rename\_goal(call, rec\_call\_subst)$
let C_{before} be the sequence $C_{1,j}$ where $(1, j) \in before$

redefine p as $p(Inv, In, Out)$:-
    $switch($
        $\langle C_{before}, base\_call\rangle,$
        $\langle C_{2,1} \ldots C_{2,n}\rangle$
).

for each $(1, j) \in assoc$
    $R_{1,j} = rename(CS_{1,j}, assoc\_call\_subst)$
for each $(1, j) \in construct$
    $R_{1,j} = rename(C_{1,j}, acc\_var\_subst)$
for $i = 1$ to n
    $B_{2,i} = rename\_goal(C_{2,i}, head\_to\_call)$
for each $(1, j) \in assoc \cup construct$
    $B_{1,j} = rename\_goal(C_{1,j}, acc\_var\_subst)$

$Acc = rename\_vars(init\_accs, acc\_var\_subst)$
let R_{assoc} be the sequence $R_{1,j}$ where $(1, j) \in assoc$
let $R_{construct}$ be the sequence $R_{1,j}$ where $(1, j) \in construct$
let B_{after} be the sequence $B_{1,j}$ where $(1, j) \in assoc \cup construct$

define p_acc as
    $p\_acc(Inv, In, Out, Acc)$:-
        $switch($
            $\langle C_{before}, R_{assoc}, rec\_call, R_{construct}\rangle,$
            $\langle B_{2,1} \ldots B_{2,n}, B_{after}\rangle$
).

Figure 3: The third stage